

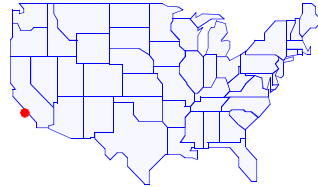
DESIGN BY CONTRACT

and the Component Revolution

Bertrand Meyer
Interactive Software Engineering & Monash University

Author's address:

Interactive Software Engineering
ISE Building, 270 Storke Road
Santa Barbara, CA 93117 USA
Telephone 805-685-1006, Fax 805-685-6869
E-mail <info@eiffel.com> <http://tools.com>



<http://eiffel.com>

© Bertrand Meyer, 1988-1999

CONT 99-9

DESIGN BY CONTRACT

1

PLAN

• Introduction, references, terminology	3
• Part 1: Issues	22
• Part 2: Principles	36
• Part 3: Applications	74
Part 3.1: What are contracts good for?	75
Part 3.2: Contracts and quality assurance	75
Part 3.3: Contracts and documentation	86
Part 3.4: Contracts and inheritance	98
Part 3.5: Handling abnormal cases	108
Part 3.6: An example project	124
Part 3.7: Methodological notes	130
Part 3.8: Other contract constructs	141
• Part 4: Tools	147
• Part 5: Sources and further developments	172

CONT 99-9

DESIGN BY CONTRACT

2

PART 0:

INTRODUCTION AND OVERVIEW

CONT 99-9

DESIGN BY CONTRACT

3

DESIGN BY CONTRACT

A systematic method for making software reliable.

Applications:

- Better analysis and design.
- Implementation.
- Testing, debugging, quality assurance.
- Documentation.
- Basis for exception handling.
- Controlling inheritance (sub-contracting).
- Project management: preserving top designers' work.
- Built-in reliability.
(Harlan Mills, 1975: "*How to write correct programs, and know it*".)

CONT 99-9

DESIGN BY CONTRACT

4

DESIGN BY CONTRACT: SCOPE

Methodological principles are language- and tool-independent.

Applications to debugging, quality assurance, testing, documentation, exception handling, inheritance require language and tool support:

- Built-in in the Eiffel language, the BON analysis & design method & notation, and the supporting tools (EiffelBench, EiffelCase). See also Catalysis and OCL.
- Can be partially emulated in C++ through macros.
- Various proposed extensions for Java.
- Extensions proposed for other languages.

See part 4.

CONT 99-9

DESIGN BY CONTRACT

5

DESIGN BY CONTRACT

Every software element is intended to satisfy a certain goal, for the benefit of other software elements (and ultimately of human users).

This goal is the element's **contract**.

The contract of any software element should be

- Explicit.
- Part of the software element itself.

CONT 99-9

DESIGN BY CONTRACT

6

A NEW VIEW OF SOFTWARE CONSTRUCTION

Constructing systems as structured collections of cooperating software elements — **clients** and **suppliers** — cooperating on the basis of clear definitions of **obligations** and **benefits**.

These definitions are the contracts.

CONT 99-9

DESIGN BY CONTRACT

7

PROPERTIES OF CONTRACTS

A contract:

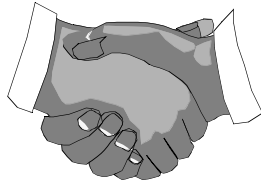
- Binds two parties (or more): client, supplier.
- Is explicit (written).
- Specifies mutual obligations and benefits.
- Usually maps obligation for one of the parties into benefit for the other, and conversely.
- Has **no hidden clauses**: obligations are those specified.
- Often relies, implicitly or explicitly, on general rules applicable to all contracts (laws, regulations, standard practices).

CONT 99-9

DESIGN BY CONTRACT

8

A HUMAN CONTRACT



deliver	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Bring package before 4 PM; pay fee.	(From postcondition:) Get package delivered by 10 AM next day.
Supplier	(Satisfy postcondition:) Deliver package by 10 AM next day.	(From precondition:) Not required to do anything if package delivered after 4 PM, or fee not paid.

CONT 99-9

DESIGN BY CONTRACT

9

AN ANALYSIS CONTRACT

```

deferred class PLANE inherit
  AIRCRAFT
feature
  start_take_off is
    -- Initiate take-off procedures.
    require
      controls.passed; assigned_runway.clear
    deferred
    ensure
      assigned_runway.owner = Current
      moving
    end
    start_landing, increase_altitude, decrease_altitude,
    moving, altitude, speed, time_since_take_off
    ... [Other features]
  invariant
    (time_since_take_off <= 120) implies (assigned_runway.owner = Current)
    moving = (speed > 10)
  end

```

Annotations:

- Precondition:** points to `controls.passed; assigned_runway.clear`
- Postcondition:** points to `assigned_runway.owner = Current`
- Class invariant:** points to `(time_since_take_off <= 120) implies (assigned_runway.owner = Current)`
- Deferred:** points to `deferred` keyword
- Not implemented:** points to `-- i.e. specified only -- not implemented`

CONT 99-9

DESIGN BY CONTRACT

10

AN ANALYSIS CONTRACT

```

deferred class VAT inherit
  TANK
feature
  in_valve, out_valve: VALVE
  fill is
    -- Fill the vat.
    require
      in_valve.open; out_valve.closed
    deferred
    ensure
      in_valve.closed; out_valve.closed; is_full
    end
    empty, is_full, is_empty, gauge, maximum,
    ...[Other features] ...
  invariant
    is_full = ((gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum))
  end
end

```

Annotations:

- Precondition:** points to `in_valve.open; out_valve.closed`
- Postcondition:** points to `in_valve.closed; out_valve.closed; is_full`
- Class invariant:** points to `is_full = ((gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum))`
- Deferred:** points to `deferred` keyword
- Not implemented:** points to `-- i.e. specified only -- not implemented`

CONT 99-9

DESIGN BY CONTRACT

11

RUNWAYS

```

deferred class RUNWAY feature
  owner: AIRCRAFT
  clear: BOOLEAN
  ... [Other features] ...
  invariant
    clear_iff_owned: not clear = (owner /= Void)
  end
end

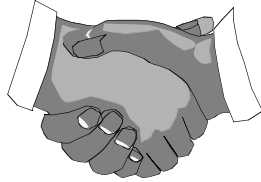
```

CONT 99-9

DESIGN BY CONTRACT

12

CONTRACTS FOR ANALYSIS



fill	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure input valve is open, output valve is closed.	(From postcondition:) Get filled-up vat, with both valves closed.
Supplier	(Satisfy postcondition:) Fill the vat and close both valves.	(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position.

CONT 99-9

DESIGN BY CONTRACT

13

CONVENTIONS AND TERMINOLOGY

Assumed: object-oriented software construction. Modular unit is the class. A class also describes a type.

Classes introduce **features**.

Features are of two kinds:

- **Attributes** (data members, instance variables), representing fields of the corresponding objects.
- **Routines** (methods, subprograms), representing algorithms. Routines include **procedures** and **functions**.

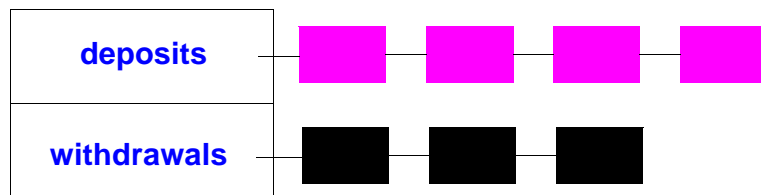
Inheritance — single, multiple and repeated.

CONT 99-9

DESIGN BY CONTRACT

14

ATTRIBUTES AND ROUTINES



Attributes: **deposits**, **withdrawals**

Routine: **balance**

CONT 99-9

DESIGN BY CONTRACT

15

CONVENTIONS AND TERMINOLOGY

A class or feature may be **deferred** (abstract, pure virtual): specified, but not implemented (or not fully implemented).

A non-deferred class or feature is **effective**.

A deferred class may include effective features.

A class may redeclare an inherited features:

- **Redefinition** (change implementation)
- **Effecting** (provide implementation if original was deferred)

CONT 99-9

DESIGN BY CONTRACT

16

DESIGN PRINCIPLES

Information hiding

Data abstraction; access to objects is only through official interface. (Violated in Java, C++; see page 153).

Two relations between classes: client, inheritance.

Uniform access principle (see next)

CONT 99-9

DESIGN BY CONTRACT

17

THE PRINCIPLE OF UNIFORM ACCESS

The features of a class must be accessed by clients in the same way whether implemented by computation or by storage.

(Definition: A client of a module is any module that uses its services.)

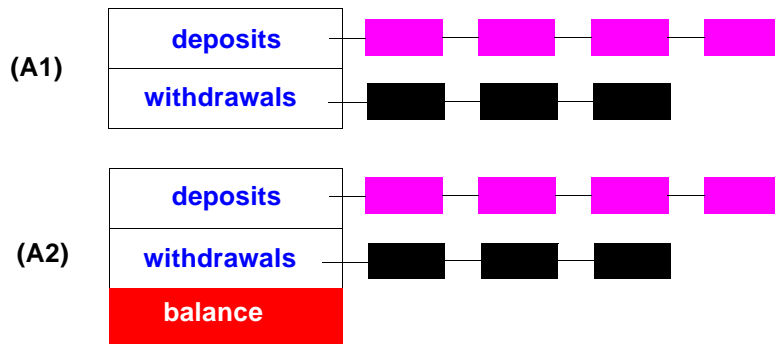
Notational issue only; but has a potentially big impact in large, long-running developments.

CONT 99-9

DESIGN BY CONTRACT

18

UNIFORM ACCESS: AN EXAMPLE — BANK ACCOUNTS



Consistency constraint:

$$\text{balance} = \text{deposits.total} - \text{withdrawals.total}$$

Ada, Pascal, C:

`a ■ balance`
`balance (a)`

Simula, Eiffel, ...:

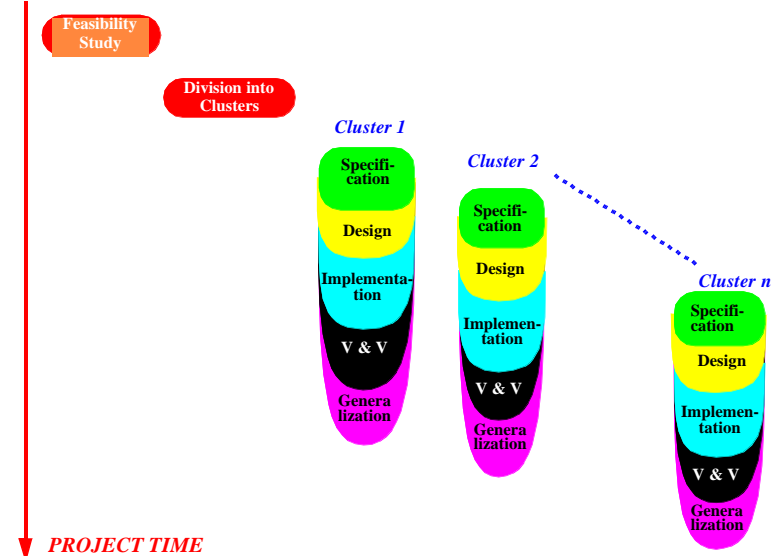
`a ■ balance`

CONT 99-9

DESIGN BY CONTRACT

19

INCREMENTAL DEVELOPMENT: THE CLUSTER MODEL

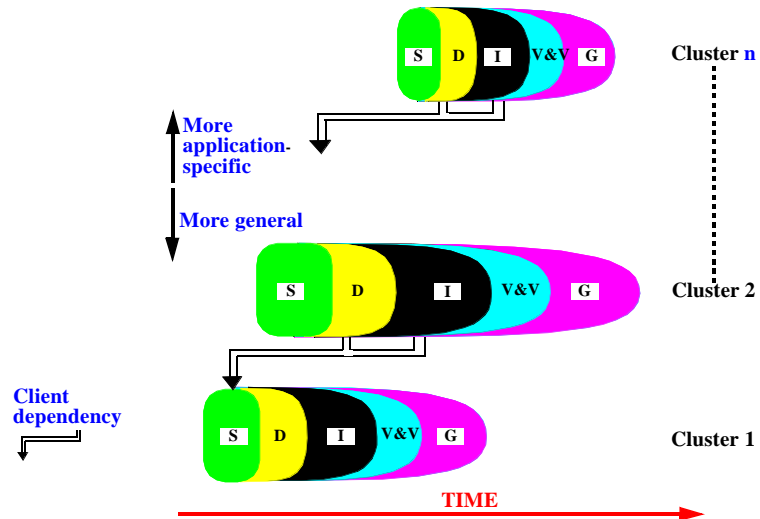


CONT 99-9

DESIGN BY CONTRACT

20

CLUSTER RELATIONS



CONT 99-9

DESIGN BY CONTRACT

21

PART 1:

ISSUES

CONT 99-9

DESIGN BY CONTRACT

22

THE GOAL: SOFTWARE RELIABILITY

Software quality factors (partial list):

- RELIABILITY
- REUSABILITY
- EXTENDIBILITY
- PORTABILITY
- EASE OF LEARNING
- EASE OF OPERATION
- INTEGRITY
- ... (cf. OOSC-2)

CONT 99-9

DESIGN BY CONTRACT

23

COMPONENTS OF RELIABILITY



Correctness:

- The ability of a software system to perform according to the specification, in cases defined by the specification.

Robustness:

- The ability of a software system to react in a reasonable manner to cases not covered by the specification.

CONT 99-9

DESIGN BY CONTRACT

24

SOFTWARE QUALITY

Currently not the principal concern of decision makers.

The weakest link in the software industry.

CONT 99-9

DESIGN BY CONTRACT

25

NON-QUALITY

Source: (Standish Group) Ted Lewis, IEEE *Computer*, July 1998

31% of 175,000 projects surveyed canceled before completion.

\$81 billion in damaged goods

52% of remaining projects ran over budget by an average of 189% (**\$59 billion** in 1995).

9% of projects on time and under budget.

Total estimate for 1998: **\$365 billion** = Microsoft + Intel + Cisco

CONT 99-9

DESIGN BY CONTRACT

26

NON-QUALITY

September 1997: missile Cruiser **USS Yorktown**, “dead in the water” for two hours and 45 minutes, due to a divide by zero in Windows NT.

Ariane 5 ESA rocket launcher (see Jean-Marc Jézéquel & BM, IEEE *Computer*, January 1997)

Therac-25

London Ambulance System

Year 2000 (see Christopher Creele, BM and Philippe Stephan, IEEE *Computer*, November 1997)

CONT 99-9

DESIGN BY CONTRACT

27

APPROACHES TO QUALITY

A posteriori (testing)

“Test, test and retest”

CONT 99-9

DESIGN BY CONTRACT

28

APPROACHES TO QUALITY (TECHNICAL)

Formal specification and verification

- Fully formal: Z (Abrial/Oxford), Object Z (U. Queensland), VDM, OBJ (SRI), Larch (MIT), B (Abrial), VSE (Germany) ...
- Partly formal: Design by Contract

Programming language support

- Static typing
- Garbage collection
- No pointer arithmetic, gotos etc.
- Object technology: abstraction, structure, information hiding
- Clear, simple syntax

Style standards

CONT 99-9

DESIGN BY CONTRACT

29

APPROACHES TO QUALITY (MANAGERIAL)

- Capability Maturity Model (Software Engineering Institute).
- ISO 9001
- Buy from market leader
- Get software in source form, benefit from public scrutiny
- Metrics collection and application
- Code reviews

CONT 99-9

DESIGN BY CONTRACT

30

APPROACHES TO QUALITY: COMPONENTS

Reuse, components, COTS (Commercial Off-The-Shelf), CBD (Component-Based Development).

Component experience:

- O-O libraries: Smalltalk, Eiffel, STL, ...
- Binary components

Binary component standards:

- CORBA
- COM/DCOM
- Enterprise Java Beans

The major component issue:

Component quality

CONT 99-9

DESIGN BY CONTRACT

31

ARIANE 5, JUNE 1996

\$10s of billions.

40 seconds into flight, exception in Ada program not handled; order given to abort the mission.

Exception was caused by an incorrect conversion: a 64-bit real value was incorrectly translated into a 16-bit integer.

- Not a design error.
- Not an implementation error.
- Not a language issue.
- Not really a testing problem.
- Only partly a quality assurance issue.

Systematic analysis had PROVED that the exception could not occur — the 64-bit value (“horizontal bias” of the flight) was proved to be always representable as a 16-bit integer!

CONT 99-9

DESIGN BY CONTRACT

32

THE ARIANE-5 FAILURE (CONTINUED)

It was a REUSE error:

- The analysis was correct — for Ariane 4!
- The assumption was documented — in a design document!

With Design by Contract, the error would almost certainly (if not avoided in the first place) detected by either static inspection or testing:

integer_bias (b: REAL): INTEGER is

require

representable (b)

do

...

ensure

equivalent (b, Result)

end

CONT 99-9

DESIGN BY CONTRACT

33

THE ARIANE-5 FAILURE (CONCLUSION)

The main lesson:

Reuse without a contract is sheer folly

See:

Jean-Marc Jézéquel and Bertrand Meyer

Design by Contract: The Lessons of Ariane

IEEE Computer, January 1997.

Also at <http://eiffel.com>

CONT 99-9

DESIGN BY CONTRACT

34

QUALITY: THE ROAD TOWARDS A SOLUTION

No “just do this” approach.

- Component-based development
- Formal or partially formal techniques (Design by Contract)
- Object technology
- Modern programming language techniques
- Systematic testing
- Open source
- Systematic metrics collection and analysis
- Management, engineering process etc.

CONT 99-9

DESIGN BY CONTRACT

35

PART 2:

PRINCIPLES

CONT 99-9

DESIGN BY CONTRACT

36

CORRECTNESS IN SOFTWARE

Correctness is a relative notion: consistency of implementation vis-à-vis specification. (This assumes there is a specification!)

Basic notation: (P , Q : assertions, i.e. properties of the state of the computation. A : instructions).

$$\{P\} A \{Q\}$$

“Hoare triple” (after C.A.R. (“Tony”) Hoare, Oxford University).

What this means (total correctness):

Any execution of A started in a state satisfying P will terminate in a state satisfying Q .

CONT 99-9

DESIGN BY CONTRACT

37

SPECIFYING A SQUARE ROOT ROUTINE

 $\{x \geq 0\}$
 $y := \text{sqrt}(x)$
 $\{\text{abs}(y^2 - x) \leq 2 * \text{epsilon} * y\}$

CONT 99-9

DESIGN BY CONTRACT

39

HOARE TRIPLES: A SIMPLE EXAMPLE

$$\{n > 5\} \quad n := n + 9 \quad \{n > 13\}$$

Most interesting properties:

- *Strongest* postcondition (from given precondition).
- *Weakest* precondition (from given postcondition).

“ P is stronger than or equal to Q ” means:

P implies Q

QUIZ: What is the strongest possible assertion? The weakest?

CONT 99-9

DESIGN BY CONTRACT

38

SOFTWARE CORRECTNESS: A QUIZ

Consider

$$\{P\} A \{Q\}$$

Take this as a job ad in the classifieds.

Should a lazy employment candidate hope for a weak or strong P ? What about Q ?

Two special offers:

- 1. $\{False\} A \{...\}$
- 2. $\{...\} A \{True\}$
-

CONT 99-9

DESIGN BY CONTRACT

40

SPECIFYING A SQUARE ROOT ROUTINE

```
{x >= 0}
```

```
y := sqrt (x)
```

```
{abs (y ^ 2 - x) <= 2 * epsilon * y}
```

CONT 99-9

DESIGN BY CONTRACT

41

A CONTRACT (FROM EIFFELBASE)

```
extend (new: G; key: H)
```

```
-- Assuming there is no item of key key,  
-- insert new with key; set inserted.
```

```
require
```

```
not_key_present: not has (key)
```

```
ensure
```

```
insertion_done: item (key) = new
```

```
key_present: has (key)
```

```
inserted: inserted
```

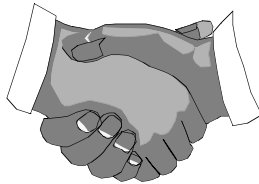
```
one_more: count = old count + 1
```

CONT 99-9

DESIGN BY CONTRACT

42

THE CONTRACT



	OBLIGATIONS	BENEFITS
Client	PRECONDITION	POSTCONDITION
Supplier	POSTCONDITION	PRECONDITION

CONT 99-9

DESIGN BY CONTRACT

43

A CLASS WITHOUT CONTRACTS

```
class ACCOUNT feature
```

```
-- Balance and minimum balance:
```

```
balance: INTEGER
```

```
Minimum_balance: INTEGER is 1000
```

```
feature {NONE} -- Implementation of deposit and withdrawal
```

```
add (sum: INTEGER) is
```

```
-- Add sum to the balance (secret procedure).
```

```
do
```

```
balance := balance + sum
```

```
end
```

CONT 99-9

DESIGN BY CONTRACT

44

WITHOUT CONTRACTS (CONTINUED)

feature -- Deposit and withdrawal operations

deposit (sum: INTEGER) is
 -- Deposit sum into the account.

do
 add (sum)
end

withdraw (sum: INTEGER) is
 -- Withdraw sum from the account.

do
 add (-sum)
end

may_withdraw (sum: INTEGER): BOOLEAN is
 -- Is it permitted to withdraw sum from the account?

do
 Result := (balance >= Minimum_balance + sum)
end

end -- class ACCOUNT

CONT 99-9

DESIGN BY CONTRACT

45

INTRODUCING CONTRACTS

class ACCOUNT create

make

feature -- Initialization

make (initial_amount: INTEGER) is
 -- Set up account with initial_amount

require
 large_enough: initial_amount >= Minimum_balance

do
 balance := initial_amount
end

feature -- Balance and minimum balance

balance: INTEGER
 Minimum_balance: INTEGER is 1000

CONT 99-9

DESIGN BY CONTRACT

46

INTRODUCING CONTRACTS (CONTINUED)

feature {NONE} -- Implementation of deposit and withdrawal

add (sum: INTEGER) is
 -- Add sum to the balance (secret procedure).

do
 balance := balance + sum
end

CONT 99-9

DESIGN BY CONTRACT

47

WITH CONTRACTS (CONTINUED)

feature -- Deposit and withdrawal operations

deposit (sum: INTEGER) is
 -- Deposit sum into the account

require
 not_too_small: sum >= 0

do
 add (sum)

ensure
 increased: balance = old balance + sum

end

CONT 99-9

DESIGN BY CONTRACT

48

WITH CONTRACTS (CONTINUED)

withdraw (sum: INTEGER) is

-- Withdraw sum from the account

require

not_too_small: sum >= 0

not_too_big: sum <= balance – Minimum_balance

do

add (–sum) -- i.e. balance := balance – sum

ensure

decreased: balance = old balance – sum

end

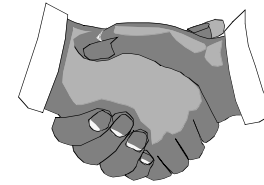
About old see also Nana, page 154

CONT 99-9

DESIGN BY CONTRACT

49

THE CONTRACT



withdraw

OBLIGATIONS

BENEFITS

Client

(Satisfy precondition:)
Make sure sum is neither too small nor too big.

(From postcondition:)
Get account updated with sum withdrawn.

Supplier

(Satisfy postcondition:)
Update account for withdrawal of sum.

(From precondition:)
Simpler processing: may assume that sum is within allowable bounds

CONT 99-9

DESIGN BY CONTRACT

50

THE IMPERATIVE AND THE APPLICATIVE

do balance := balance – sum	ensure balance = old balance – sum
PRESCRIPTIVE	DESCRIPTIVE
How	What
Operational	Denotational
Implementation	Specification
Command	Query
Instruction	Expression
Imperative	Applicative

CONT 99-9

DESIGN BY CONTRACT

51

WITH CONTRACTS (END)

may_withdraw (sum: INTEGER): BOOLEAN is

-- Is it possible to withdraw sum from the account?

do

Result := (balance >= Minimum_balance + sum)

end

invariant

not_under_minimum: balance >= Minimum_balance

end -- class ACCOUNT

CONT 99-9

DESIGN BY CONTRACT

52

THE CLASS INVARIANT

Consistency constraint applicable to all instances of a class.

Must be satisfied:

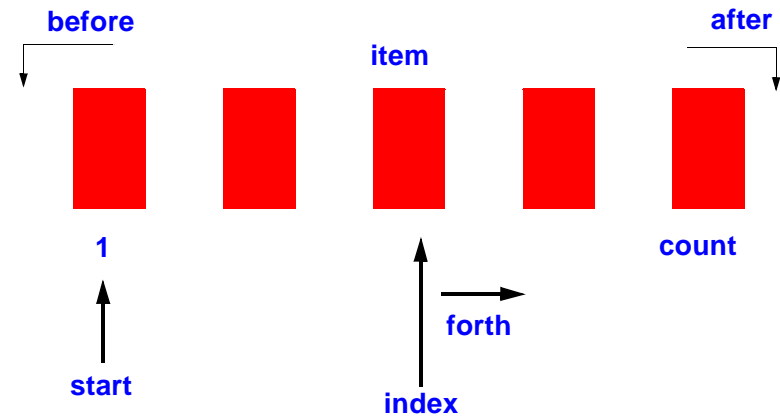
- After creation.
- After execution of any feature by any client.
(Qualified calls only: **a.f (...)**)

CONT 99-9

DESIGN BY CONTRACT

53

LIST STRUCTURES

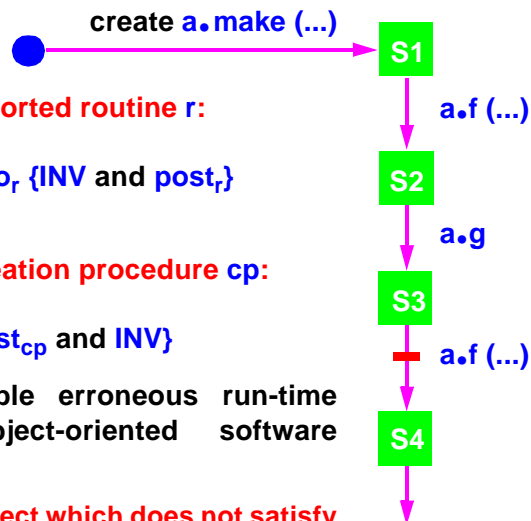


CONT 99-9

DESIGN BY CONTRACT

54

THE CORRECTNESS OF A CLASS



(1-*n*) For every exported routine *r*:

{INV and *pre_r*} *do_r* {INV and *post_r*}

(1-*m*) For every creation procedure *cp*:

{*pre_{cp}*} *do_{cp}* {*post_{cp}* and INV}

The worst possible erroneous run-time situation in object-oriented software development:

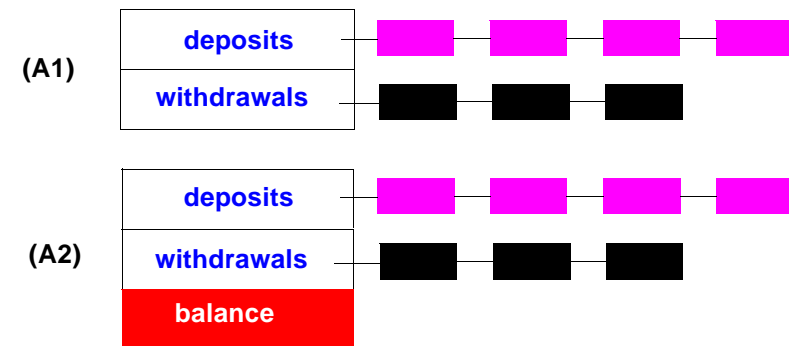
- Producing an object which does not satisfy the invariant of its own class.

CONT 99-9

DESIGN BY CONTRACT

55

UNIFORM ACCESS: AN EXAMPLE — BANK ACCOUNTS



balance = deposits.total – withdrawals.total

CONT 99-9

DESIGN BY CONTRACT

56

A MORE SOPHISTICATED VERSION

```
class ACCOUNT create
```

```
    make
```

```
feature {NONE} -- Implementation
```

```
    add (sum: INTEGER) is
```

```
        -- Add sum to the balance (secret procedure).
```

```
    do
```

```
        balance := balance + sum
```

```
    end
```

```
deposits: DEPOSIT_LIST
```

```
withdrawals: WITHDRAWAL_LIST
```

CONT 99-9

DESIGN BY CONTRACT

57

NEW VERSION (CONTINUED)

```
feature {NONE} -- Initialization
```

```
    make (initial_amount: INTEGER) is
```

```
        -- Set up account with initial_amount
```

```
    require
```

```
        large_enough: initial_amount >= Minimum_balance
```

```
    do
```

```
        balance := initial_amount
```

```
        create deposits.make
```

```
        create withdrawals.make
```

```
    end
```

```
feature -- Balance and minimum balance
```

```
    balance: INTEGER
```

```
    Minimum_balance: INTEGER is 1000
```

CONT 99-9

DESIGN BY CONTRACT

58

NEW VERSION (CONTINUED)

```
feature -- Deposit and withdrawal operations
```

```
    deposit (sum: INTEGER) is
```

```
        -- Deposit sum into the account
```

```
    require
```

```
        not_too_small: sum >= 0
```

```
    do
```

```
        add (sum)
```

```
        deposits.extend (create {DEPOSIT} .make (sum))
```

```
    ensure
```

```
        increased: balance = old balance + sum
```

```
    end
```

CONT 99-9

DESIGN BY CONTRACT

59

NEW VERSION (CONTINUED)

```
    withdraw (sum: INTEGER) is
```

```
        -- Withdraw sum from the account
```

```
    require
```

```
        not_too_small: sum >= 0
```

```
        not_too_big: sum <= balance - Minimum_balance
```

```
    do
```

```
        add (-sum)
```

```
        withdrawals.extend (create {WITHDRAWAL} .make (sum))
```

```
    ensure
```

```
        decreased: balance = old balance - sum
```

```
    end
```

CONT 99-9

DESIGN BY CONTRACT

60

NEW VERSION (END)

```

may_withdraw (sum: INTEGER): BOOLEAN is
  -- Is it possible to withdraw sum from the account?
  do
    Result := (balance >= Minimum_balance + sum)
  end
invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total
end -- class ACCOUNT

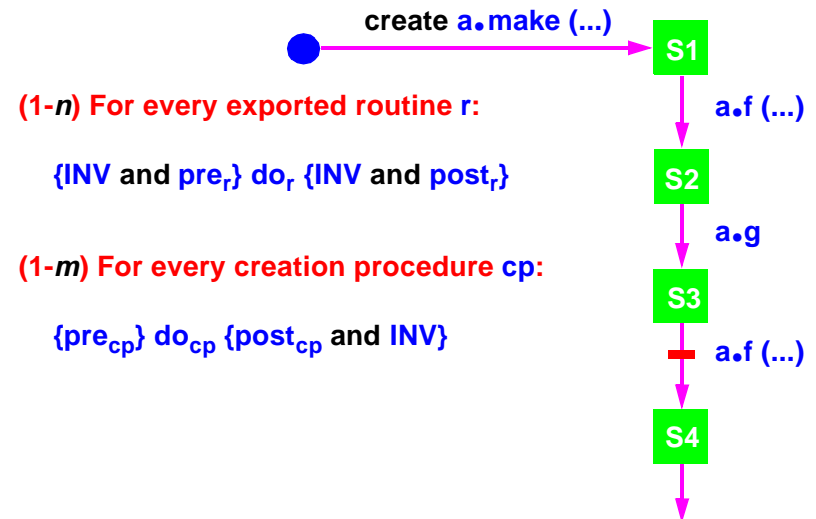
```

CONT 99-9

DESIGN BY CONTRACT

61

THE CORRECTNESS OF A CLASS

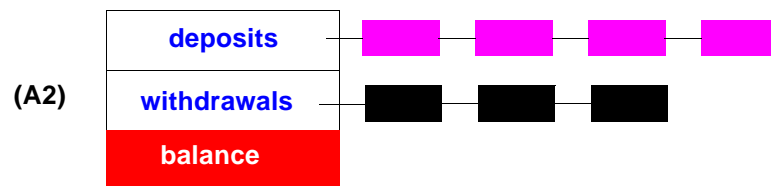


CONT 99-9

DESIGN BY CONTRACT

62

SECOND BANK ACCOUNT REPRESENTATION



balance = deposits.total - withdrawals.total

CONT 99-9

DESIGN BY CONTRACT

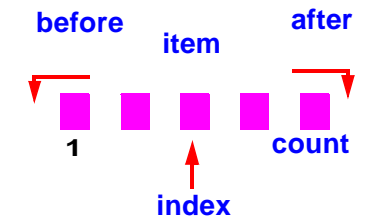
63

USING CONTRACTS

```

class ARRAYED_LIST [G] inherit
  LIST
  redefine put_left, put_right end
  ARRAY
  rename put as array_put end
feature
  put_right (x: G) is
    -- Insert x right of cursor position; move cursor to new item.
    require
      not after
    do
      ...
    ensure
      item = x
      count = old count + 1
      index = old index + 1
    end
    ... Other features ...
  invariant
    0 <= index; index <= count + 1
end -- class ARRAYED_LIST

```

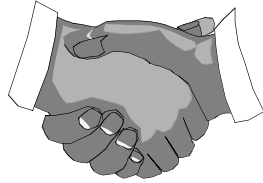


CONT 99-9

DESIGN BY CONTRACT

64

DESIGN BY CONTRACT



put_right	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure cursor is at valid insertion position.	(From postcondition:) Get x added to the list at desired position.
Supplier	(Satisfy postcondition:) Insert x to right of cursor position; bring cursor to new item.	(From precondition:) Simpler processing thanks to assumption that cursor is valid insertion position.

CONT 99-9

DESIGN BY CONTRACT

65

OBJECT-ORIENTED ANALYSIS USING CONTRACTS

deferred class **VAT** inherit**TANK**

feature

in_valve, out_valve: **VALVE**

fill is

-- Fill the vat.

require

in_valve.open; out_valve.closed

deferred

ensure

in_valve.closed; out_valve.closed; is_full

end

empty, is_full, is_empty, gauge, maximum,
... [Other features] ...

invariant

is_full = (gauge >= 0.97 * maximum) and (gauge <= 1.03 * maximum)

end

Precondition

-- i.e. specified only
-- not implemented

Postcondition

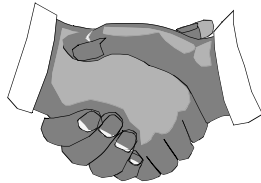
Class invariant

CONT 99-9

DESIGN BY CONTRACT

66

CONTRACTS FOR ANALYSIS



fill	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Make sure input valve is open, output valve is closed.	(From postcondition:) Get filled-up vat, with both valves closed.
Supplier	(Satisfy postcondition:) Fill the vat and close both valves.	(From precondition:) Simpler processing thanks to assumption that valves are in the proper initial position.

CONT 99-9

DESIGN BY CONTRACT

67

FURTHER EXAMPLES

See e.g. **LIST**, **LINKED_LIST** in EiffelBase

Pay particular attention to class invariants.

CONT 99-9

DESIGN BY CONTRACT

68

CONTRACTS: RUN-TIME EFFECT

Compilation options (per class, in Eiffel):

- 1• No assertion checking
- 2• Preconditions only
- 3• Preconditions and postconditions
- 4• Preconditions, postconditions, class invariants
- 5• All assertions

For the difference between levels 4 and 5, see page 141.

CONT 99-9

DESIGN BY CONTRACT

69

A CONTRACT VIOLATION IS NOT A SPECIAL CASE

For special cases

(e.g. “if the sum is negative, report an error...”)

use standard control structures, e.g. if ... then ... else.

A run-time assertion violation is something else: the manifestation of

A DEFECT (“BUG”)

CONT 99-9

DESIGN BY CONTRACT

70

THE CONTRACT LANGUAGE

Language of boolean expressions (plus old):

- No predicate calculus (i.e. no quantifiers, \forall or \exists).
- Function calls permitted, e.g. (in a **STACK** class):

<p>put (x: G) is -- Push x on top of stack require not full do ... ensure not empty end</p>	<p>remove is -- Pop top of stack require not empty do ... ensure not full end</p>
--	--

CONT 99-9

DESIGN BY CONTRACT

71

THE CONTRACT LANGUAGE

First order predicate calculus may be desirable, but not sufficient anyway.

Example: “The graph has no cycles”.

In assertions, use only side-effect-free functions.

Use of iterators provides the equivalent of first-order predicate calculus in connection with a library such as EiffelBase or STL. For example:

`my_integer_list.for_all (~ is_positive (?))`

with

`is_positive (x: INTEGER): BOOLEAN is do Result := (x > 0) end`

CONT 99-9

DESIGN BY CONTRACT

72

THE IMPERATIVE AND THE APPLICATIVE

do balance := balance – sum	ensure balance = old balance – sum
PRESCRIPTIVE	DESCRIPTIVE
How	What
Operational	“Denotational”
Implementation	Specification
Command	Query
Instruction	Expression
Imperative	“Applicative”

CONT 99-9

DESIGN BY CONTRACT

73

PART 3:

APPLICATIONS

CONT 99-9

DESIGN BY CONTRACT

74

PART 3.1: WHAT ARE CONTRACTS GOOD FOR?

- Writing correct software (analysis, design, implementation, maintenance, reengineering).
 - Documentation (the “short” form of a class).
 - Effective reuse.
 - Controlling inheritance.
 - Preserving the work of the best developers.
-
- Quality assurance, testing, debugging.
(especially in connection with the use of libraries)
 - Exception handling

CONT 99-9

DESIGN BY CONTRACT

75

A CONTRACT VIOLATION IS NOT A SPECIAL CASE

For special cases

(e.g. “if the sum is negative, report an error...”)

use standard control structures, e.g. if ... then ... else.

A run-time contract violation is something else: the manifestation of

A DEFECT (“BUG”)

CONT 99-9

DESIGN BY CONTRACT

76

PART 3.2: CONTRACTS AND QUALITY ASSURANCE

Precondition violation: **BUG IN THE CLIENT.**

Postcondition violation: **BUG IN THE SUPPLIER.**

Invariant violation: **BUG IN THE SUPPLIER.**

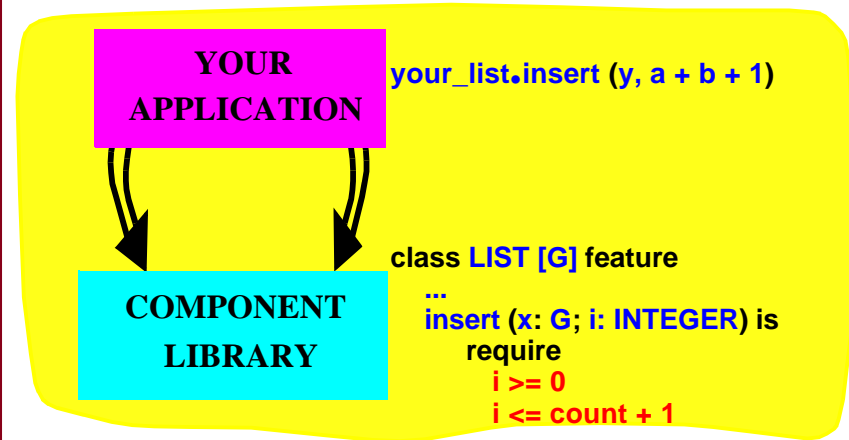
CONT 99-9

DESIGN BY CONTRACT

77

CONTRACTS AND BUG TYPES

Preconditions are particularly useful to find bugs in **client** code:



CONT 99-9

DESIGN BY CONTRACT

78

CONTRACTS AND QUALITY ASSURANCE

Use run-time assertion monitoring for quality assurance, testing, debugging.

Compilation options (reminder):

- 1• No assertion checking
- 2• Preconditions only
- 3• Preconditions and postconditions
- 4• Preconditions, postconditions, class invariants
- 5• All assertions

For the difference between levels 4 and 5, see page 141.

CONT 99-9

DESIGN BY CONTRACT

79

CONTRACTS AND QUALITY ASSURANCE

Contracts enable QA activities to be based on a precise description of what they expect.

Profoundly transform the activities of testing, debugging and maintenance.

CONT 99-9

DESIGN BY CONTRACT

80

DEBUGGING WITH CONTRACTS: AN EXAMPLE

This example will use a live demo from ISE's EiffelBench, with a "planted" error leading to a precondition violation.

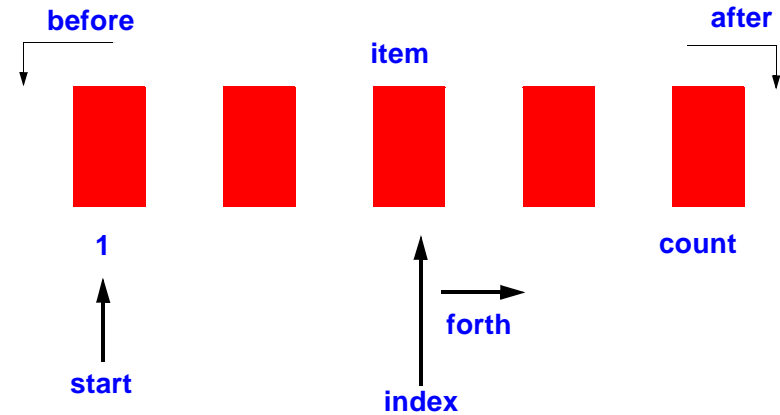
The example uses both the browsing and debugging mechanisms.

CONT 99-9

DESIGN BY CONTRACT

81

TO UNDERSTAND THE EXAMPLE: LIST CONVENTIONS

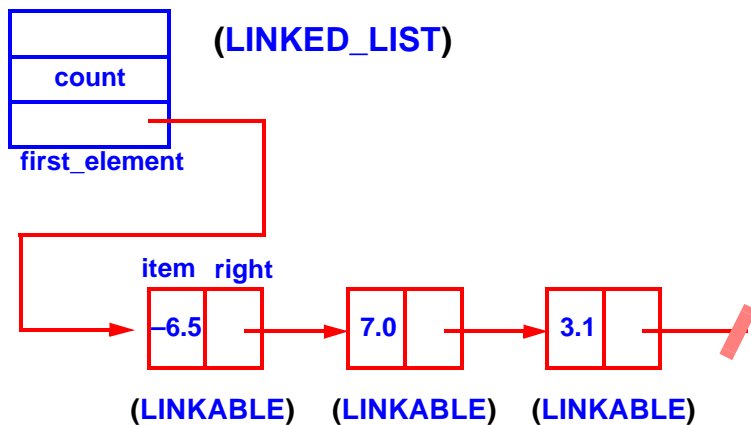


CONT 99-9

DESIGN BY CONTRACT

82

LINKED LIST REPRESENTATION



CONT 99-9

DESIGN BY CONTRACT

83

ADDING AND CATCHING A BUG

In class **STARTER**, procedure **make_a_list**, replace the first call to **extend** by a call to **put**.

Execute system. What happens?

Use browsing mechanisms to find out what's wrong (violated precondition).

To understand, consider what the diagram of page 82 becomes when the number of list items goes to zero.

CONT 99-9

DESIGN BY CONTRACT

84

CONTRACT MONITORING

Enabled or disabled by compile-time options.

Default: preconditions only.

In development: use “all assertions” whenever possible.

During operation: normally, should disable monitoring. But have an assertion-monitoring version ready for shipping.

Result of an assertion violation: exception.

Ideally: static checking (proofs) rather than dynamic monitoring.

CONT 99-9

DESIGN BY CONTRACT

85

PART 3.3: CONTRACTS AND DOCUMENTATION

Recall example class:

class **ACCOUNT** create

make

feature {NONE} -- **Implementation**

add (sum: INTEGER) is

-- Add sum to the balance (secret procedure).

do

balance := balance + sum

end

deposits: **DEPOSIT_LIST**

withdrawals: **WITHDRAWAL_LIST**

CONT 99-9

DESIGN BY CONTRACT

86

CLASS EXAMPLE (CONTINUED)

feature -- **Initialization**

make (initial_amount: INTEGER) is

-- Set up account with initial_amount

require

large_enough: initial_amount >= Minimum_balance

do

balance := initial_amount

create deposits.make

create withdrawals.make

end

feature -- **Balance and minimum balance**

balance: INTEGER

Minimum_balance: INTEGER is 1000

CONT 99-9

DESIGN BY CONTRACT

87

CLASS EXAMPLE (CONTINUED)

feature -- **Deposit and withdrawal operations**

deposit (sum: INTEGER) is

-- Deposit sum into the account

require

not_too_small: sum >= 0

do

add (sum)

deposits.extend (create {DEPOSIT} .make (sum))

ensure

increased: balance = old balance + sum

end

CONT 99-9

DESIGN BY CONTRACT

88

CLASS EXAMPLE (CONTINUED)

```

withdraw (sum: INTEGER) is
  -- Withdraw sum from the account
  require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Minimum_balance
  do
    add (-sum)
    withdrawals.extend(create {WITHDRAWAL} .make (sum))
  ensure
    decreased: balance = old balance - sum
  end

```

CONT 99-9

DESIGN BY CONTRACT

89

CLASS EXAMPLE (END)

```

may_withdraw (sum: INTEGER): BOOLEAN is
  -- Is it possible to withdraw sum from the account?
  do
    Result := (balance >= Minimum_balance + sum)
  end
invariant
  not_under_minimum: balance >= Minimum_balance
  consistent: balance = deposits.total - withdrawals.total
end -- class ACCOUNT

```

CONT 99-9

DESIGN BY CONTRACT

90

SHORT FORM: DEFINITION

See also: Javadoc

Simplified form of class text, retaining interface elements only:

- Remove any non-exported (private) feature.

For the exported (public) features:

- Remove body (do clause).
- Keep header comment if present.
- Keep contracts: preconditions, postconditions, class invariant.
- Remove any contract clause that refers to a secret feature.
(This raises a problem; can you see it?)

CONT 99-9

DESIGN BY CONTRACT

91

EXPORT RULE FOR PRECONDITIONS

In

```

feature {A, B, C}
  r (...) is
    require
      some_property
    ...
  end

```

some_property must be exported (at least) to **A**, **B** and **C**!

No such requirement for postconditions and invariants.

CONT 99-9

DESIGN BY CONTRACT

92

SHORT FORM OF ACCOUNT CLASS

```

class interface ACCOUNT create
  make (initial_amount: INTEGER)
    -- Set up account with initial_amount.
  require
    initial_amount >= 0

  feature

    balance: INTEGER

    Minimum_balance: INTEGER is 1000

    deposit (sum: INTEGER)
      -- Deposit sum into account.
    require
      not_too_small: sum >= 0
    ensure
      increased: balance = old balance + sum

```

CONT 99-9

DESIGN BY CONTRACT

93

SHORT FORM (CONTINUED)

```

withdraw (sum: INTEGER)
  -- Withdraw sum from account.
  require
    not_too_small: sum >= 0
    not_too_big: sum <= balance - Minimum_balance
  ensure
    decreased: balance = old balance - sum

  may_withdraw (sum: INTEGER): BOOLEAN
    -- Is it permitted to withdraw sum from the account?

  invariant

    not_under_minimum: balance >= Minimum_balance

end -- class interface ACCOUNT

```

CONT 99-9

DESIGN BY CONTRACT

94

FLAT, FLAT-SHORT

Flat form of a class: reconstructed class with all the features at the same level (immediate and inherited). Takes renaming, redefinition etc. into account.

The flat-form is an inheritance-free client-equivalent form of the class.

Flat-short form: the short form of the flat form. Full interface documentation.

CONT 99-9

DESIGN BY CONTRACT

95

USES OF THE (FLAT-)SHORT FORM

- Documentation, manuals
- Design
- Communication between developers
- Communication between developers and managers

CONT 99-9

DESIGN BY CONTRACT

96

CONTRACTS AND REUSE

The short form — i.e. the set of contracts governing a class — should be the standard form of library documentation.

Reuse without a contract is sheer folly.

See the Ariane 5 example.

CONT 99-9

DESIGN BY CONTRACT

97

PART 3.4: CONTRACTS AND INHERITANCE

THE INVARIANT RULE

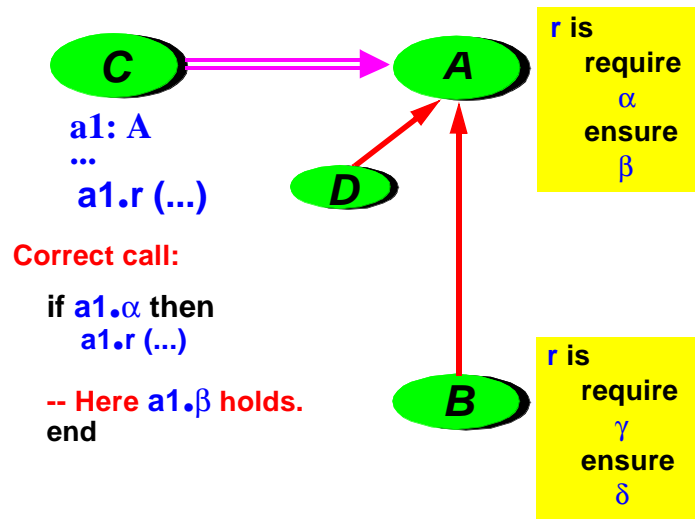
The invariant of a class automatically includes the invariant clauses from all its parents, “and”-ed.

CONT 99-9

DESIGN BY CONTRACT

98

CONTRACTS AND INHERITANCE



CONT 99-9

DESIGN BY CONTRACT

99

ASSERTION REDECLARATION RULE

When redeclaring a routine:

- Precondition may only be kept or weakened.
- Postcondition may only be kept or strengthened.

Redeclaration covers both redefinition and effecting.

CONT 99-9

DESIGN BY CONTRACT

100

ASSERTION REDECLARATION RULE (EIFFEL)

Redeclared version may **not** have **require** or **ensure**.

May have nothing (assertions kept by default), or

```
require else new_pre
ensure then new_post
```

Resulting assertions are:

original_precondition or **new_pre**

original_postcondition and **new_post**

CONT 99-9

DESIGN BY CONTRACT

101

DON'T CALL US, WE'LL CALL YOU

deferred class **LIST [G]** inherit

CHAIN [G]

feature

has (x: G): **BOOLEAN** is

-- Does x appear in list?

do

from **start** until **after** or else **found (x)** loop

forth

end

Result := not **after**

end

CONT 99-9

DESIGN BY CONTRACT

102

SEQUENTIAL STRUCTURES (Continued)

forth is

require

not **after**

deferred

ensure

index = old **index** + 1

end

start is

deferred

ensure

empty or else **position** = 1

end

CONT 99-9

DESIGN BY CONTRACT

103

SEQUENTIAL STRUCTURES

position: **INTEGER** is deferred end

... **empty**, **found**, **after**, ...

invariant

0 <= **position**; **position** <= **size** + 1

empty implies (**after** or **before**)

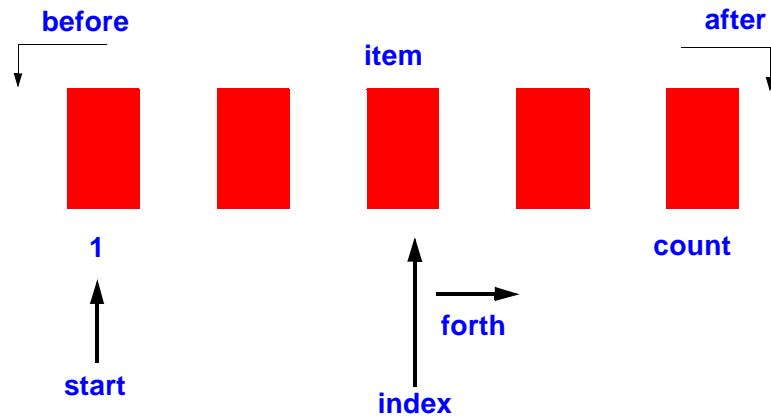
end -- class **LIST**

CONT 99-9

DESIGN BY CONTRACT

104

SEQUENTIAL STRUCTURES



CONT 99-9

DESIGN BY CONTRACT

105

IMPLEMENTATION VARIANTS

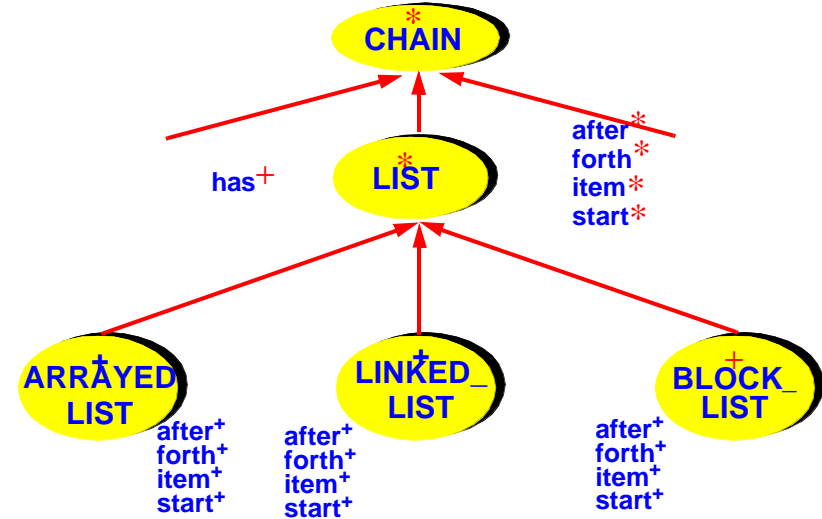
	start	forth	after	found (x)
Arrayed list	$i := 1$	$i := i + 1$	$i > \text{count}$	$t @ i = x$
Linked list	$c := \text{first_cell}$	$c := c.\text{right}$	$c = \text{Void}$	$c.\text{item} = x$
File	rewind	read	end_of_file	$f \uparrow = x$

CONT 99-9

DESIGN BY CONTRACT

107

DESCENDANT IMPLEMENTATIONS



CONT 99-9

DESIGN BY CONTRACT

106

PART 3.5: EXCEPTION HANDLING

The need for exceptions arises when the contract is broken.

Two concepts:

- **Failure**: a routine, or other operation, is unable to fulfill its contract.
- **Exception**: an undesirable event occurs during the execution of a routine — as a result of the **failure** of some operation called by the routine.

CONT 99-9

DESIGN BY CONTRACT

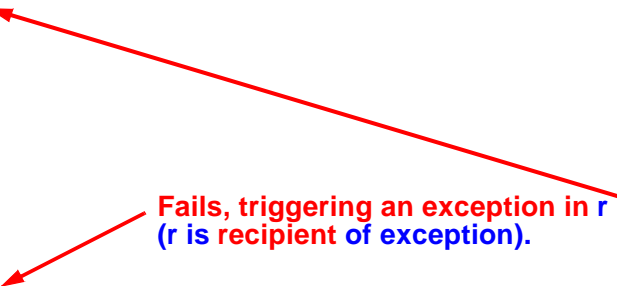
108

THE ORIGINAL STRATEGY

```

r (...) is
do
  op1
  op2
  ...
  opi
  ...
  opn
end

```



CONT 99-9

DESIGN BY CONTRACT

109

CAUSES OF EXCEPTIONS

Assertion violation

Void call (**x.f** with no object attached to **x**)

Operating system signal (arithmetic overflow, no more memory, interrupt ...)

CONT 99-9

DESIGN BY CONTRACT

110

THE EXCEPTION HANDLING RULE

SAFE EXCEPTION HANDLING PRINCIPLE

There are only two acceptable ways to react for the recipient of an exception:

- Try again, using a different strategy (or repeating the same strategy) (*Retrying*).
- Concede failure, and trigger an exception in the caller (*Organized Panic*).

CONT 99-9

DESIGN BY CONTRACT

111

HOW NOT TO DO IT

(From an Ada textbook)

```

sqrt (x: REAL) return REAL is
begin
  if x < 0.0 then
    raise Negative;
  else
    normal_square_root_computation;
  end
exception
  when Negative =>
    put ("Negative argument");
    return;
  when others => ...
end; -- sqrt

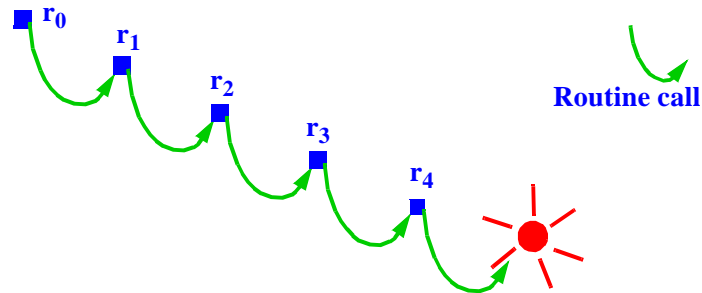
```

CONT 99-9

DESIGN BY CONTRACT

112

THE CALL CHAIN



CONT 99-9

DESIGN BY CONTRACT

113

EXCEPTION MECHANISM

Two constructs:

- A routine may contain a **rescue** clause.
- A rescue clause may contain a **retry** instruction.

A **rescue** clause that does not execute a **retry** leads to failure of the routine (this is the organized panic case).

CONT 99-9

DESIGN BY CONTRACT

114

TRANSMITTING OVER AN UNRELIABLE LINE (1)

Max_attempts: INTEGER is 100

attempt_transmission (message: STRING) is

-- Transmit message in at most Max_attempts attempts.

local

failures: INTEGER

do

unsafe_transmit (message)

rescue

failures := failures + 1

if failures < Max_attempts then

retry

end

end

CONT 99-9

DESIGN BY CONTRACT

115

TRANSMITTING OVER AN UNRELIABLE LINE (2)

Max_attempts: INTEGER is 100

failed: BOOLEAN

attempt_transmission (message: STRING) is

-- Try to transmit message; if impossible in at most

-- Max_attempts, set failed to true.

local

failures: INTEGER

do

if failures < Max_attempts then

unsafe_transmit (message)

else

failed := True

end

rescue

failures := failures + 1

retry

end

CONT 99-9

DESIGN BY CONTRACT

116

IF NO EXCEPTION CLAUSE (1)

Absence of a rescue clause is equivalent, in first approximation, to an empty rescue clause:

```
f (...) is
do
...
end
```

is an abbreviation for

```
f (...) is
do
...
rescue
  -- Nothing here
end
```

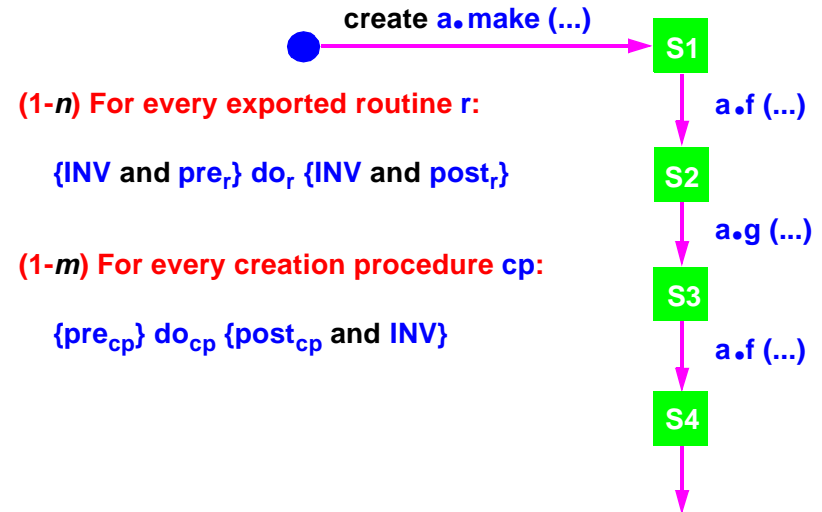
(This is a provisional rule; see page 122.)

CONT 99-9

DESIGN BY CONTRACT

117

THE CORRECTNESS OF A CLASS



CONT 99-9

DESIGN BY CONTRACT

118

EXCEPTION CORRECTNESS: A QUIZ

For the normal body:

$\{INV \text{ and } pre_r\} do_r \{INV \text{ and } post_r\}$

For the exception clause:

$\{ ??? \} \quad rescue_r \quad \{ ??? \}$

CONT 99-9

DESIGN BY CONTRACT

119

QUIZ ANSWERS

For the normal body:

$\{INV \text{ and } pre_r\} do_r \{INV \text{ and } post_r\}$

For the rescue clause:

$\{ True \} \quad rescue_r \quad \{ INV \}$

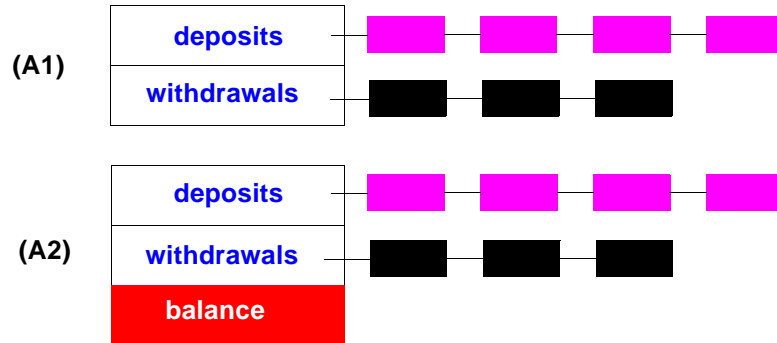
CONT 99-9

DESIGN BY CONTRACT

120

UNIFORM ACCESS: AN EXAMPLE — BANK ACCOUNTS

balance = deposits.total – withdrawals.total



CONT 99-9

DESIGN BY CONTRACT

121

IF NO EXCEPTION CLAUSE (2)

Absence of a rescue clause is equivalent to a default rescue clause:

```
f (...) is
do
...
end
```

is an abbreviation for

```
f (...) is
do
...
rescue
    default_rescue
end
```

The task of default_rescue is to restore the invariant.

CONT 99-9

DESIGN BY CONTRACT

122

FOR FINER-GRAIN EXCEPTION HANDLING

Use class **EXCEPTIONS** from the Kernel Library.

Some features:

exception (code of last exception that was triggered).

is_assertion_violation, etc.

raise ("EX_NAME")

CONT 99-9

DESIGN BY CONTRACT

123

PART 3.6: AN EXAMPLE PROJECT

Laser printer software at Hewlett-Packard

1997-1998

Embedded system development: software runs on chip in printer

Host development environment: VxWorks operating system

About 800,000 lines of legacy C code.

CONT 99-9

DESIGN BY CONTRACT

124

INTRODUCING DESIGN BY CONTRACT

First in C and C++ through macros.

Eiffel introduced later, in particular because of memory management requirements. C calls Eiffel (CECIL library).

CONT 99-9

DESIGN BY CONTRACT

125

BENEFITS

Greatly decreased error rates in the elements built with Design by Contract.

Several major errors found in the legacy C code.

Bug in chip.

CONT 99-9

DESIGN BY CONTRACT

126

CONTRACTS AS A SAFEGUARD FOR SOFTWARE EVOLUTION

Christopher Creel (HP project leader):

“In most companies today, you have a small group of hard guns who are responsible for the core job.

But later the other engineers come in, and because they don't immediately understand the solution they start hacking it, and in the process destroy it.”

“The consequence for the quality of the code base is a *lowest common denominator* effect: the quality degrades to the level of the work of those who are not as good.”

CONT 99-9

DESIGN BY CONTRACT

127

DESIGN BY CONTRACT TO THE RESCUE.

“Design by Contract addresses this. The original designers build a white-box framework: a scaffold to which you will plug in the working components — the implementations.

Because they can see further into the future than most engineers, they puts the contracts in place for all eternity, destroying errors that other engineers could make 2 years or 10 years later because they don't see the whole picture.”

CONT 99-9

DESIGN BY CONTRACT

128

FURTHER TECHNICAL ISSUES

- Using contracts well: methodological notes
- Invariants and business rules.
- Checking invariants: why before *and* after?
- Other contract constructs.

CONT 99-9

DESIGN BY CONTRACT

129

A NOTE ON PRECONDITIONS

The client must **guarantee** the precondition before the call.

This does not necessarily mean **testing** for the precondition.

Schema 1 (testing):

```
if not my_stack.full then
  my_stack.put (some_element)
end
```

Schema 2 (guaranteeing without testing):

```
my_stack.remove
...
my_stack.put (some_element)
```

CONT 99-9

DESIGN BY CONTRACT

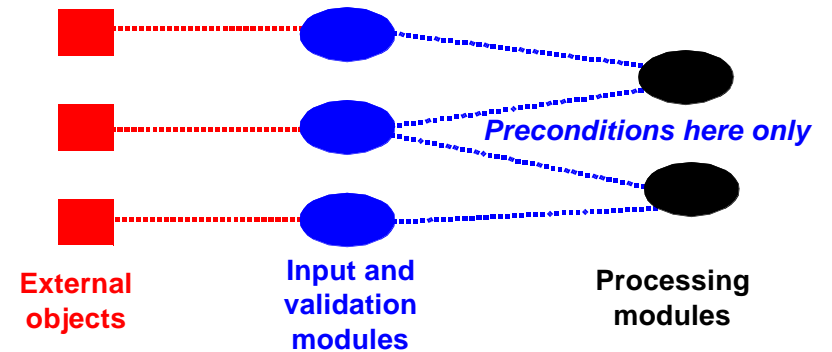
131

PART 3.7: METHODOLOGICAL NOTES

CONTRACTS ARE NOT INPUT CHECKING TESTS...

... but they can be used to help weed out undesirable input.

Filter modules:



CONT 99-9

DESIGN BY CONTRACT

130

ANOTHER EXAMPLE

`sqrt (x, epsilon: REAL): REAL` is

-- Square root of x, precision epsilon

require

```
x >= 0
epsilon >= 10 ^ (-6)
```

do

...

ensure

```
abs (Result ^ 2 - x) <= 2 * epsilon * Result
```

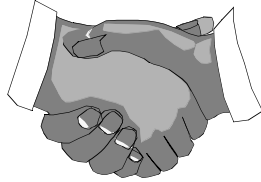
end

CONT 99-9

DESIGN BY CONTRACT

132

THE CONTRACT



sqrt	OBLIGATIONS	BENEFITS
Client	(Satisfy precondition:) Provide non-negative value and requested precision that is not too small	(From postcondition:) Get square root within requested precision.
Supplier	(Satisfy postcondition:) Produce square root within requested precision	(From precondition:) Simpler processing thanks to assumptions on value and precision

CONT 99-9

DESIGN BY CONTRACT

133

NOT DEFENSIVE PROGRAMMING

For every consistency condition that is required to perform a certain operation:

- Assign responsibility for the condition to one of the contract's two parties (client, supplier).
- Stick to this decision: do not duplicate responsibility.

Simplifies software and improves global reliability.

CONT 99-9

DESIGN BY CONTRACT

135

NOT DEFENSIVE PROGRAMMING!

It is never acceptable to have a routine of the form

```

sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
    x >= 0
    epsilon >= 10 ^ (-6)
  do
    if x < 0 then
      ... Do something about it (?) ...
    else
      ... Normal square root computation ...
    end
  ensure
    abs (Result ^ 2 - x) <= 2 * epsilon * Result
end

```

CONT 99-9

DESIGN BY CONTRACT

134

HOW STRONG SHOULD A PRECONDITION BE?

Two opposite styles:

- **TOLERANT:** weak preconditions (including the weakest, *True*: no precondition).
- **DEMANDING:** strong preconditions, requiring the client to make sure all logically necessary conditions are satisfied before each call.

Partly a matter of taste.

But: demanding style leads to a better distribution of roles, provided the precondition is:

- Justifiable in terms of the specification only.
- Documented (through the short form).
- Reasonable!

CONT 99-9

DESIGN BY CONTRACT

136

A DEMANDING STYLE

```

sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  -- (same version as before)
  require
    x >= 0
    epsilon >= 10 ^ (-6)
  do
    ...
  ensure
    abs (Result ^ 2 - x) <= 2 * epsilon * Result
  end

```

CONT 99-9

DESIGN BY CONTRACT

137

A TOLERANT STYLE

```

sqrt (x, epsilon: REAL): REAL is
  -- Square root of x, precision epsilon
  require
    True
  do
    if x < 0 then
      ... Do something about it (?) ...
    else
      ... Normal square root computation ...
    end
  ensure
    computed implies
      abs (Result ^ 2 - x) <= 2 * epsilon * Result
  end

```



CONT 99-9

DESIGN BY CONTRACT

138

CONTRASTING STYLES

```

put (x: G) is
  -- Push x on top of stack.
  require
    not full
  do
    ....
  end
tolerant_put (x: G) is
  -- Push x if possible, otherwise set impossible to true.
  do
    if not full then
      put (x)
    else
      impossible := True
    end
  end
end

```

CONT 99-9

DESIGN BY CONTRACT

139

INVARIANTS AND BUSINESS RULES

Invariants are absolute consistency conditions.

They can serve to represent business rules if knowledge is to be built into the software.

FORM 1

invariant

not_under_minimum: balance >= Minimum_balance

FORM 2

invariant

not_under_minimum_if_normal:
normal_state implies (balance >= Minimum_balance)

CONT 99-9

DESIGN BY CONTRACT

140

PART 3.8: OTHER CONTRACT CONSTRUCTS

The “check” instruction

```
check
  ... Assertion ...
end
```

Often used in a call to a precondition-equipped routine:

```
x := a ^2 + b^2
... Other instructions ...
check
  x >= 0
  -- Because x was computed above
  -- as a sum of squares.
end
y := sqrt (x)
```

CONT 99-9

DESIGN BY CONTRACT

141

OTHER ASSERTION CONSTRUCTS: LOOP INVARIANTS AND VARIANTS

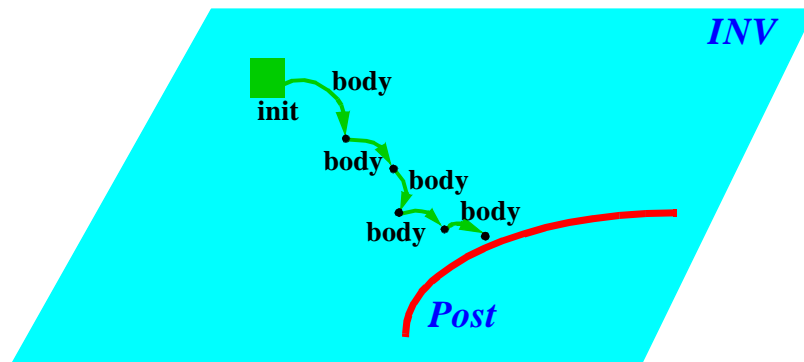
```
from
  x := a; y := b
invariant
  x > 0; y > 0
  -- The pair <x, y> has the same greatest common divisor
  -- as the pair <a, b>
variant
  x + max (y)
until
  x = y
loop
  if x > y then x := x - y else y := y - x end
end
```

CONT 99-9

DESIGN BY CONTRACT

142

LOOPS AS COMPUTATIONS BY APPROXIMATION

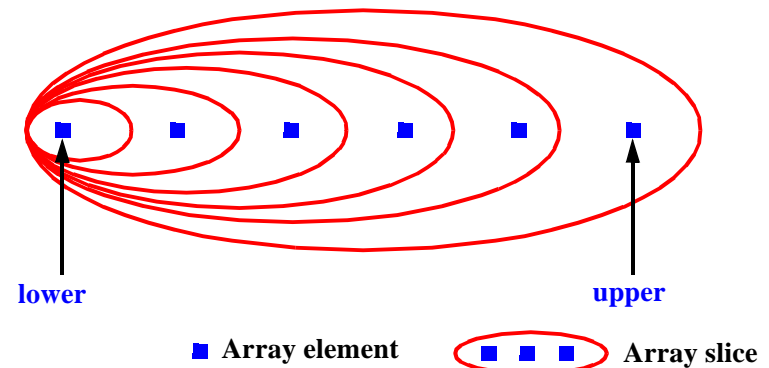


CONT 99-9

DESIGN BY CONTRACT

143

AN ARRAY COMPUTATION



```
from i := t.lower ; Result := t @ lower until i = t.upper loop
  i := i + 1; Result := Result.max (t @ i)
end
```

CONT 99-9

DESIGN BY CONTRACT

144

EXERCISE

Using a loop invariant and a loop variant, produce a correct and formally justified version of binary search in a sorted array.

(See wrong versions on following page.)

CONT 99-9

DESIGN BY CONTRACT

145

PART 4:**TOOLS**

CONT 99-9

DESIGN BY CONTRACT

147

BINARY SEARCH: ERRONEOUS VERSIONS

BS1	BS2	BS3	BS4
<pre> from i := 1; j := n until i = j loop m := (i + j) // 2 if t @ m <= x then i := m else j := m end end Result := (x = t @ i) </pre>	<pre> from i := 1; j := n found := False until i = j and not found loop m := (i + j) // 2 if t @ m < x then i := m + 1 elseif t @ m = x then found := True else j := m - 1 end end Result := found </pre>	<pre> from i := 0; j := n until i = j loop m := (i + j + 1) // 2 if t @ m <= x then i := m + 1 else j := m end end if i >= 1 and i <= n then Result := (x = t @ i) else Result := False end </pre>	<pre> from i := 0; j := n + 1 until i = j loop m := (i + j) // 2 if t @ m <= x then i := m + 1 else j := m end end if i >= 1 and i <= n then Result := (x = t @ i) else Result := False end </pre>

CONT 99-9

DESIGN BY CONTRACT

146

**DESIGN BY CONTRACT AND
LANGUAGES/METHODS**

Eiffel, Sather: built-in

Contract extensions have been proposed for Ada 83, Smalltalk, C++, Java, Python...

In analysis and design methods/tools:

- Built-in in BON (Business Object Notation).
- UML extension: Object Constraint Language

CONT 99-9

DESIGN BY CONTRACT

148

THE BUSINESS OBJECT NOTATION

KIM WALDÉN, JEAN-MARC NERSON

SYSTEMATIC O-O ANALYSIS

BON provides a clear notation and methodological guidelines for high-level analysis and design. Three key concepts: **seamlessness**, **reversibility** and **software contracting**.

- Well-defined set of conventions.
- Supports semantics (contracts, ...), not just structure.
- Mechanisms for systematic development; supports Design by Contract.
- Textual as well as graphical variants. Three views: graphics (bubbles and arrows!), tables, formal text (Eiffel-like).
- Meant for use with software tools (EiffelCase)

Scales up: abstraction and grouping facilities: Classes, Clusters, entire systems. Zoom in, zoom out, abstract.

CONT 99-9

DESIGN BY CONTRACT

149

BON REFERENCE

Seamless Object-Oriented Software Architecture

Kim Waldén and Jean-Marc Nerson

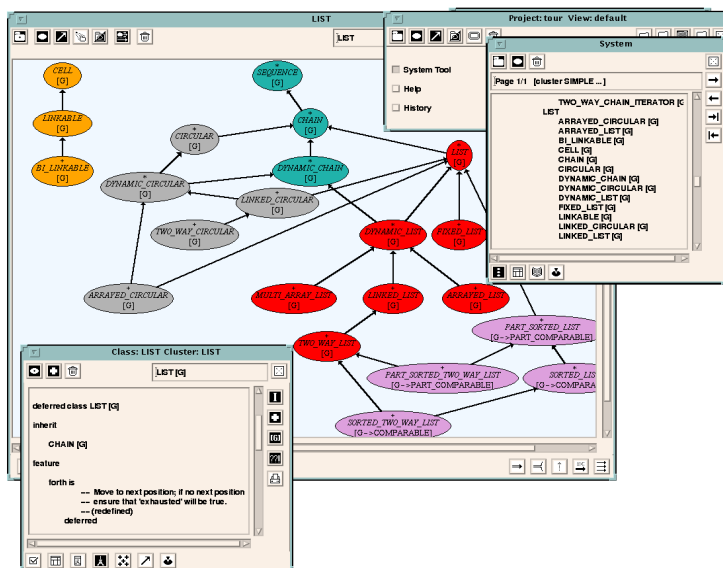
Prentice Hall, 1995

CONT 99-9

DESIGN BY CONTRACT

150

EIFFELCASE: REENGINEERING A LIBRARY



CONT 99-9

DESIGN BY CONTRACT

151

APPLYING DESIGN BY CONTRACT IN NON-EIFFEL ENVIRONMENTS

Basic step: use standardized comments, or graphical annotations, corresponding to require, ensure, invariant clauses.

In programming languages:

- Macros
- Preprocessor

Use of macros avoids the trouble of preprocessors, but invariants are more difficult to handle than preconditions and postconditions.

Difficulties: contract inheritance; “short”-like tools; link with exception mechanism.

CONT 99-9

DESIGN BY CONTRACT

152

C++/JAVA DESIGN BY CONTRACT LIMITATIONS

The possibility of direct assignments

```
x.attrib = value
```

limits the effectiveness of contracts: circumvents the official class interface of the class. In a fully O-O language, use:

```
x.set_attrib (value)
```

with

```
set_attrib (v: TYPE) is
```

(cf p. 17)

```
-- Make v the next value for attrib.
```

```
require
```

```
... Some condition on v ...
```

```
do
```

```
attrib := v
```

```
ensure
```

```
attrib = v
```

```
end
```

CONT 99-9

DESIGN BY CONTRACT

153

C++ CONTRACTS

GNU Nana: improved support for contracts and logging in C and C++.

P.J. Maker, Australia, see:

<http://www.cs.ntu.edu.au/homepages/pjm/nana-home/>

Set of C++ macros and commands for gdb debugger. Replaces assert.h. Validated only with GCC. *“Support existed in earlier versions of Nana for the GNU Ada compiler. We may add support for Ada and FORTRAN in the future if anyone is interested.”*

Support for quantifiers (Forall, Exists, Exists1) corresponding to iterations on the STL (C++ Standard Template Library).

Support for time-related contracts (“Function must execute in less than 1000 cycles”).

CONT 99-9

DESIGN BY CONTRACT

154

NANA EXAMPLE

```
void qsort(int v[], int n) { /* sort v[0..n-1] */
  DI(v != NULL && n >= 0); /* check arguments under gdb(1) only */
  L("qsort(%p, %d)\n", v, n); /* log messages to a circular buffer */
  ...; /* the sorting code */
}
```

```
  I(A(int i = 1, i < n, i++, /* verify v[] sorted (Forall) */
    v[i-1] <= v[i])); /* forall i in 1..n-1 @ v[i-1] <= v[i] */
}
```

```
void intsqr(int &r) { /* r' = floor(sqrt(r)) */
  DS($r = r); /* save r away into $r for later use under gdb(1) */
  DS($start = $cycles); /* real time constraints */
  ...; /* code which changes r */
  DI($cycles - $start < 1000); /* code must take less than 1000 cycles */
  DI(((r * r) <= $r) && ($r < (r + 1) * (r + 1))); /* use $r in postcondition */
}
```

Back to page 49

CONT 99-9

DESIGN BY CONTRACT

155

NANA

In the basic package: no real notion of class invariant. (“Invariant”, macro DI, is equivalent of “check” instruction.)

Package eiffel.h *“is intended to provide a similar setup to Eiffel in the C++ language. It is a pretty poor emulation, but it is hopefully better than nothing.”*

Macros: CHECK_NO, CHECK_REQUIRE, CHECK_ENSURE, CHECK_INVARIANT, CHECK_LOOP, CHECK_ALL.

Using CHECK_INVARIANT assumes a boolean-valued class method called invariant. Called only if a REQUIRE or ENSURE clause is present in the method.

No support for contract inheritance.

CONT 99-9

DESIGN BY CONTRACT

156

NANA EIFFEL.H EXAMPLE

(Source: Nana Web page.)

```
#include <eiffel.h>

class example {
  int nobjects;
  map<location,string,locationIt> layer;
public:
  bool invariant();
  void changeit(location l);
};

bool example::invariant() {
  return AO(i,layer,valid_location((*i).first)) && nobjects >= 0;
}
```

CONT 99-9

DESIGN BY CONTRACT

157

```
void example::changeit(string n, location l) {
  REQUIRE(E1O(i,layer,(*i).second == n));
  ...;

  while(..) {
    INVARIANT(...);
    ...
    INVARIANT(...);
  }

  ...

  CHECK(x == 5);
  ...
  ENSURE(layer[l] == n);
}
```

CONT 99-9

DESIGN BY CONTRACT

158

DESIGN BY CONTRACT IN JAVA**Notes:**

- OAK 0.5 (pre-Java) contained an assertion mechanism, which was removed due to “lack of time”.
- “No assertions” is currently #4 on the Java users’ bug list.
- Several different proposals.

CONT 99-9

DESIGN BY CONTRACT

159

iContract

Reference: **iContract, the Java Design by Contract Tool**, TOOLS USA 1998, IEEE Computer Press, pages 295-307.

Java preprocessor. Assertions are embedded in special comment tags, so iContract code remains valid Java code in case the preprocessor is not available.

Support for Object Constraint Language mechanisms.

Support for assertion inheritance.

CONT 99-9

DESIGN BY CONTRACT

160

iContract example

(See attached paper.)

```

interface Person {

    /**
     * @post return > 0
     */
    int get Age ()

    /**
     * @pre age > 0
     */
    void setAge( int age );

```

CONT 99-9

DESIGN BY CONTRACT

161

ANOTHER JAVA TOOL: JASS (JAWA)

Preprocessor. Also adds Eiffel-like exception handling. See <http://theoretica.Informatik.Uni-Oldenburg.DE/~jawa/doc.engl.html>

```

class Alpha {
    int x,y,z

    ...
    private void Div()
    throws RuntimeCheck.AssertionException {
        /* check z!=0 */
        x=y/z;
        /* rescue catch (RuntimeCheck.AssertionException e) {
            z!=1; retry
        }
    }
    ...
}

```

CONT 99-9

DESIGN BY CONTRACT

162

JASS LIST

```

package DataContainer;

public class List {
    private Linkable root;
    private Linkable cursor;
    private int pos;
    private int nb_elements;

    public boolean IsEmpty()
    throws RuntimeCheck.AssertionException {
        return (nb_elements == 0);
        /* ensure result == (nb_elements==0); nochange */
    }

    public void GoToFirstElement()
    throws RuntimeCheck.AssertionException {
        /* require !IsEmpty() */
        ...
        /* ensure nb_elements==old_nb_elements */
    }
}

```

CONT 99-9

DESIGN BY CONTRACT

163

```

public void GoToNextElement()
throws RuntimeCheck.AssertionException {
    /* require !IsEmpty() */
    ...
    /* ensure nb_elements==old_nb_elements */
}

public void GoToLastElement()
throws RuntimeCheck.AssertionException {
    /* require !IsEmpty() */
    ...
    /* ensure nb_elements==old_nb_elements;
       cursor.GetNext()==root
    */
}

public void GoToPrevElement()
throws RuntimeCheck.AssertionException {
    /* require !IsEmpty() */
    ...
    /* ensure nb_elements==old_nb_elements */
}

```

CONT 99-9

DESIGN BY CONTRACT

164

```

public void Insert( Value value )
throws RuntimeException.AssertionException {
  /** require value!=null */
  /** ensure !isEmpty() */
}

public void Delete()
throws RuntimeException.AssertionException {
  /** require !isEmpty() */
  /** nb_elements==old_nb_elements -1 */
}

/** invariant 0<=pos; pos<=nb_elements;
    pos==GetPosition(cursor);
    cursor==null | pos!=0 ;
  */
}

```

CONT 99-9

DESIGN BY CONTRACT

165

BISCOTTI

Adds assertions to Java, through modifications of the JDK 1.2 compiler.

See IEEE *Computer*, July 1999

CONT 99-9

DESIGN BY CONTRACT

166

THE OBJECT CONSTRAINT LANGUAGE

Designed by IBM and other companies as an addition to UML.

Includes support for:

- Invariants, preconditions, postconditions
- Guards (not further specified).
- Predefined types and collection types
- Associations
- Collection operations: ForAll, Exists, Iterate

Not directly intended for execution.

Jos Warmer, AW

CONT 99-9

DESIGN BY CONTRACT

167

OCL EXAMPLES

Postconditions:

```

post: result = collection->iterate
      (elem; acc : Integer = 0 | acc + 1)

```

```

post: result = collection->iterate
      ( elem; acc : Integer = 0 |
        if elem = object then acc + 1 else acc endif)

```

```

post: T.allInstances->forAll
      (elem | result->includes(elem) = set->
        includes(elem) and set2->includes(elem))

```

Collection types include Collection, Set, Bag, Sequence

CONT 99-9

DESIGN BY CONTRACT

168

CONTRACTS FOR COM AND CORBA

See: Damien Watkins: **Using Interface Definition Languages to support Path Expressions and Programming by Contract**, **TOOLS USA 1998**, IEEE Computer Press, pages 308-317.

Set of mechanisms added to IDL to include: preconditions, postconditions, class invariants.

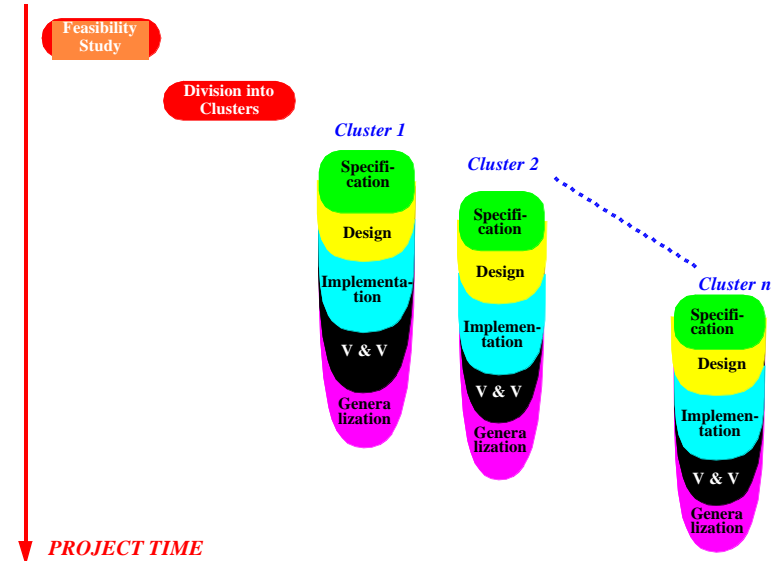
Examples: see attached paper.

CONT 99-9

DESIGN BY CONTRACT

169

THE CLUSTER MODEL OF THE SOFTWARE LIFECYCLE

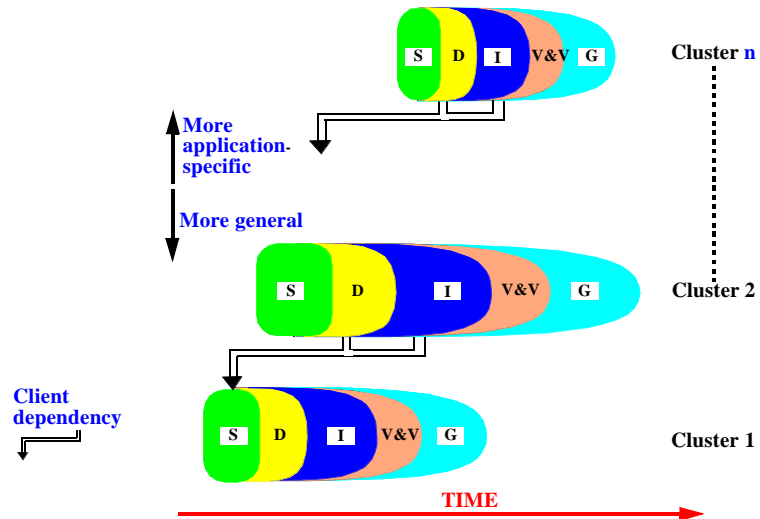


CONT 99-9

DESIGN BY CONTRACT

170

CLUSTER RELATIONS



CONT 99-9

DESIGN BY CONTRACT

171

PART 5:

**SOURCES &
FURTHER DEVELOPMENTS**

CONT 99-9

DESIGN BY CONTRACT

172

SOURCES 1*(See precise references in OOSC-2 bibliography)*

Work on formal semantics of programming languages and program proving:

- “**Assigning meaning to programs**”, Bob Floyd, 1967.
- “**An axiomatic basis for computer programming**”, C.A.R. Hoare, 1969.
- “**A Discipline of Programming**”, E.W. Dijkstra, 1976.

CONT 99-9

DESIGN BY CONTRACT

173

SOURCES 3

Work on formal specification and verification:

- **Z** language, Jean-Raymond Abrial, 1977-1980.
- **VDM**, Björner and Jones, early eighties.
- **M**, 1983.

CONT 99-9

DESIGN BY CONTRACT

175

SOURCES 2

Work on abstract data types:

- Liskov and Zilles, 1974.
- Goguen, Thatcher, Wagner, 1975.
- Guttag, 1977.
- Author, 1976.

CONT 99-9

DESIGN BY CONTRACT

174

SOURCES 4

Programming languages with assertions:

- **Algol W** (Hoare and Wirth).
- **CLU** (Liskov).
- **Euclid**, **Alphard**, **Turing**.
- **Anna** (Annotated Ada) (Luckham, Stanford University, early 80's)

CONT 99-9

DESIGN BY CONTRACT

176

FURTHER DEVELOPMENTS

More extensive contract languages (?.)

More rigorous contract languages.

Timing contracts (temporal logic).

Closer integration in analysis and design methods and tools (cf. BON, Catalysis, OCL).

Closer integration in the software process: cluster model, ISO 9001, CMM.

CONT 99-9

DESIGN BY CONTRACT

177

THE TRUSTED COMPONENTS INITIATIVE

Initiated by:
 Monash University (Melbourne)
 Interactive Software Engineering
 Univ. of Brighton, IRISA (France)
 and other institutions

MISSION

Develop the infrastructure for enabling the software industry to transform itself into a discipline based on quality reusable components.

<http://www.trusted-components.org>

CONT 99-9

DESIGN BY CONTRACT

178

THE TRUSTED COMPONENTS PROJECT**PRODUCTS**

- General-purpose components
- Special-purpose components
- Methodology of component-based developments
- Testing tools
- Proof techniques
- Procedures
- Papers, books, publications...

CONT 99-9

DESIGN BY CONTRACT

179

**A MAJOR TOOL
FOR SOFTWARE ENGINEERING**

CONT 99-9

DESIGN BY CONTRACT

180